UTN FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Backend de Aplicaciones
Apunte de Clases

Apunte 22 - Despliegue de Backend con Docker Compose

Introducción

En este apunte vamos a dar un nuevo paso: mirar al backend de nuestra aplicación como un bloque unificado. Aunque en escenarios reales el backend suele interactuar con múltiples sistemas externos, podemos definir un límite lógico que encierre los componentes propios del backend, y es sobre ese conjunto que vamos a trabajar.

Por ejemplo, elementos como el Identity Provider (en nuestro caso, Keycloak), la base de datos (como PostgreSQL) o servicios externos (como las APIs de Google, Microsoft o AWS) no forman parte estricta del backend funcional que desarrollamos. En cambio, sí podemos considerar como parte de este bloque a:

- El API Gateway
- Los microservicios funcionales
- Otros contenedores de soporte como almacenamiento de archivos (por ejemplo, MinIO)

Hasta este punto, trabajamos cada contenedor como una unidad independiente, utilizando comandos como `docker run`. A partir de ahora, vamos a evolucionar esa práctica y aprender a orquestar todos los contenedores que conforman el backend de manera conjunta, como si fueran una única unidad de despliegue y ejecución.

Para esto, vamos a utilizar Docker Compose, una herramienta diseñada específicamente para definir, configurar y ejecutar múltiples contenedores como un solo sistema lógico.

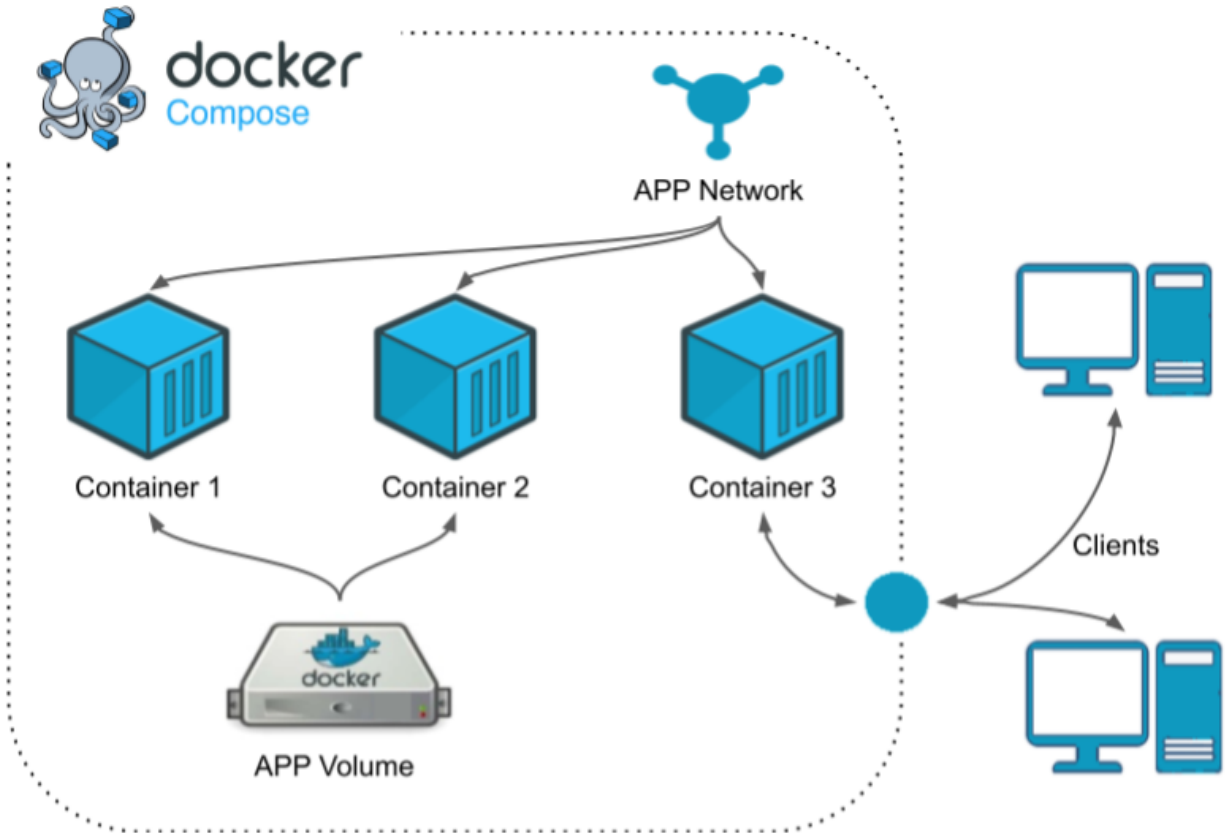
Motivación y Escenario de Uso

En etapas anteriores trabajamos con contenedores individuales, levantando una app Spring con su propia imagen Docker. Sin embargo, en la realidad del desarrollo backend moderno, nuestras aplicaciones suelen estar compuestas por **múltiples servicios que deben coexistir y comunicarse entre sí**: microservicios, bases de datos, gateways, idententy provider, etc.

Si bien podríamos levantar manualmente cada contenedor, configurar redes y volúmenes a mano, esto resulta **tedioso, propenso a errores y difícil de mantener**.

Aquí es donde entra en juego **Docker Compose**, una herramienta que nos permite **definir y orquestar** el conjunto de servicios, redes y volúmenes de una aplicación completa, en un solo archivo `docker-compose.yaml`.

La imagen que presentamos a continuación resume la idea de fondo:



- Todos los contenedores (1, 2 y 3) forman parte de una **red interna compartida** (`APP Network`), lo que permite su comunicación directa.
- Uno o varios contenedores pueden tener acceso a **volúmenes persistentes** (`APP Volume`).
- Desde el exterior, los clientes sólo pueden acceder al sistema a través de un **punto de entrada expuesto**, como podría ser un API Gateway.

Objetivos del Apunte

En este apunte vamos a:

- Comprender **qué es Docker Compose** y por qué es fundamental en entornos multicontenedor.
- Analizar **el flujo de trabajo completo** que Compose automatiza.
- Desarrollar paso a paso un archivo `docker-compose.yaml`.
- Explorar la **definición de servicios, redes y volúmenes**.
- Realizar un laboratorio integrador que utilice todos los componentes vistos hasta ahora.

Con Docker Compose **dejamos de pensar en contenedores individuales** y empezamos a pensar en **aplicaciones distribuidas**. Este cambio de perspectiva es clave para dar el siguiente paso en el dominio del desarrollo backend moderno con microservicios.

¿Qué es Docker Compose?

Docker Compose es una herramienta que nos permite definir y gestionar aplicaciones multicontenedor de forma declarativa, utilizando un archivo de configuración en formato YAML llamado `docker-compose.yaml`. Este archivo describe los servicios que componen una aplicación, incluyendo:

- Las **imágenes o Dockerfiles** a utilizar
- Los **puertos expuestos**

- Los **volúmenes** para persistencia de datos
- Las **variables de entorno**
- Las **dependencias entre servicios**
- Las **redes internas** que los conectan

Con Compose, podemos levantar toda una arquitectura de servicios con un solo comando:

```
docker compose up
```

Esto hace que Compose sea ideal para entornos de desarrollo, pruebas e incluso producción en escenarios de pequeña escala.

Diferencias entre `docker run` y Docker Compose

Aunque ambas herramientas se apoyan en el motor de Docker para crear contenedores, tienen enfoques diferentes:

Característica	**`docker run`**	**`docker compose up`**
Nivel de control	Bajo nivel (imperativo)	Alto nivel (declarativo)
Forma de uso	Ejecuta contenedores uno a uno	Define todos los servicios en un archivo YAML
Ideal para	Pruebas rápidas, comandos sueltos	Arquitecturas completas y reproducibles
Configuración	Se pasa por línea de comandos	Se define en `docker-compose.yaml`
Reusabilidad	Baja	Alta
Escalabilidad con `replicas`	Manual, con comandos separados	Integrada mediante `scale` y extensiones de Compose

Ejemplo con `docker run`:

```
docker run -d --name db -e POSTGRES_PASSWORD=admin postgres
```

Ejemplo equivalente con Compose:

A partir de lo ya compartido en el instructivo de creación de un contenedor PostgresSql

Paso 1: Crear el archivo `docker-compose.yml`

```
version: "3"
services:
```

```
  db:
    image: postgres
    environment:
      - POSTGRES_PASSWORD=admin
```

Paso 2: Levantar el contenedor con `docker compose up`

En el mismo directorio que contiene el archivo `docker-compose.yml` debemos ejecutar:

```
docker compose up
```

Con Docker Compose ganamos expresividad, mantenimiento y reproducibilidad. Es por eso que en arquitecturas de microservicios resulta imprescindible.

En las siguientes secciones vamos a analizar en detalle el formato del archivo `docker-compose.yaml`, las redes, los volúmenes y el ciclo de vida de los servicios definidos con Compose.

Esquema de Funcionamiento General de Docker Compose

Para entender cómo funciona internamente Docker Compose, partimos de un archivo llamado `docker-compose.yml`. Este archivo contiene la **descripción declarativa** de los servicios que queremos levantar, las imágenes a utilizar o construir, los volúmenes, redes, variables de entorno, puertos expuestos, y otras configuraciones clave.

¿Qué hace Docker Compose con este archivo?

Cuando ejecutamos:

```
docker compose up
```

Docker Compose realiza una serie de pasos en el siguiente orden:

1. **Lee el archivo `docker-compose.yml`** y parsea la configuración declarada.
2. **Construye las imágenes** necesarias si se declara la instrucción `build`.
3. **Descarga las imágenes** si se declara la instrucción `image` con nombre de repositorio y no están disponibles localmente.
4. **Crea una red bridge por defecto** (a menos que se declare otra) para interconectar los contenedores.
5. **Inicializa los volúmenes** definidos, ya sea anónimos, nombrados o vinculados al sistema host.
6. **Crea los contenedores** en el orden declarado respetando las dependencias (`depends_on`).
7. **Levanta los contenedores**, iniciando su proceso principal.

Una vez en ejecución, Compose mantiene los contenedores coordinados, pudiendo detenerlos y eliminarlos todos juntos con `docker compose down`.

Relación con lo ya aprendido

Hasta ahora, aprendimos a trabajar con Docker ejecutando manualmente los comandos `docker build`, `docker run`, `docker network create`, `docker volume create`, etc. Compose encapsula todas esas tareas bajo una configuración clara y repetible.

Es decir, Compose **automatiza** todo lo que hacíamos de forma manual:

Acción manual	Acción equivalente en Compose
`docker build -t miimagen .`	`build: .`
`docker run --name cont -p 9000:9000 miimagen`	`ports: ["9000:9000"]`
`docker network create app-net`	`networks:`
`docker volume create datos`	`volumes:`
`docker run --rm -v datos:/data`	`volumes:` dentro del servicio

> Podemos concluir entonces que el archivo docker-compose.yml actúa como una orquestación declarativa que le indica a Docker Compose cómo traducir nuestras definiciones en las instrucciones equivalentes de `docker run` (con todos sus parámetros de configuración) y otras operaciones de Docker que no abordamos de forma independiente por tener menor relevancia aislada, pero que Compose aplica automáticamente para asegurar el correcto funcionamiento del conjunto.

Archivo `docker-compose.yml` en detalle

En este apartado vamos a descomponer y documentar con profundidad la estructura del archivo `docker-compose.yml`, que es el núcleo del uso de Docker Compose. Este archivo nos permite definir y coordinar múltiples servicios, redes y volúmenes que componen nuestra aplicación de manera declarativa.

A continuación, se describen las principales secciones y propiedades de este archivo, con ejemplos, esquemas y aclaraciones didácticas.

1. `version`

Define la versión del esquema de configuración que se está utilizando. Aunque Docker recomienda no incluirlo desde Compose v3.9+, aún es común encontrarlo para compatibilidad.

```yaml
version: '3.9'
```

> ⚠ Nota: Si omitimos esta línea, Docker Compose usará la última versión disponible.

2. `services`

Esta es la sección principal del archivo y describe todos los contenedores que componen la aplicación. Cada entrada dentro de `services` representa un servicio y su configuración.

```yaml
services:
  gateway:
    build: ./gateway
```

```
    ports:
      - "8080:8080"
    depends_on:
      - ms-usuarios
      - ms-tarifas
  ms-usuarios:
    image: utnfc/ms-usuarios:latest
  ms-tarifas:
    image: utnfc/ms-tarifas:latest
```

En este ejemplo:

- `gateway` se construye a partir del Dockerfile en `./gateway`
- Expone el puerto `8080`
- Depende de los microservicios `ms-usuarios` y `ms-tarifas`

> [!NOTE]
>
> Cabe mencionar que en este ejemplo, las imágenes de ms-usuarios y ms-tarifas ya deben existir en el registro local o en algún registro de referencia desde donde se puedan descargar.

Propiedades comunes en `services`

Propiedad	Descripción	Ejemplo
`build`	Ruta al Dockerfile para construir la imagen localmente	`build: ./ms-usuarios`
`image`	Imagen a utilizar (desde Docker Hub o repositorio propio)	`image: postgres:16`
`container_name`	Nombre personalizado para el contenedor	`container_name: backend-gateway`
`ports`	Mapea puertos del host al contenedor	`- "8080:8080"`
`volumes`	Mapea volúmenes entre host y contenedor	`- ./data:/var/lib/postgresql/data`
`environment`	Define variables de entorno	`SPRING_PROFILES_ACTIVE=dev`
`depends_on`	Indica dependencias entre servicios	`depends_on: [db]`
`restart`	Política de reinicio del contenedor	`restart: always`

> 🧠 Tip: `depends_on` no garantiza que el servicio esté *listo*, sólo que fue iniciado. Para asegurarse que una base de datos esté operativa, es recomendable implementar healthchecks.

3. `networks`

Permite definir redes personalizadas que los servicios pueden compartir. Compose crea por defecto una red bridge si no se especifica otra, pero declarar redes explícitamente mejora la organización y la

comunicación controlada entre servicios.

```
networks:
  backend:
    driver: bridge
```

Podemos luego asignarla a cada servicio:

```
services:
  ms-usuarios:
    networks:
      - backend
```

> ✅ En la práctica, todos los servicios en una red Compose pueden referenciarse entre sí por su *nombre de servicio* como hostname.

4. `volumes`

Permite definir volúmenes persistentes que pueden ser reutilizados entre servicios o montados desde el host.

```
volumes:
  db-data:
```

Asignación en un servicio:

```
services:
  postgres:
    image: postgres:16
    volumes:
      - db-data:/var/lib/postgresql/data
```

Esto garantiza que los datos persisten incluso si el contenedor se elimina.

> 🔐 Además de persistencia, los volúmenes pueden facilitar el backup, migración y pruebas de datos.

5. `env_file`

Permite cargar un archivo `.env` con variables de entorno, facilitando la separación de configuración y código.

```
services:
  app:
```

```
    env_file:
      - .env
```

Archivo `.env`:

```
SPRING_PROFILES_ACTIVE=dev
DB_HOST=postgres
```

> ✨ Muy útil para gestionar distintos entornos como desarrollo, testeo y producción.

6. Ejemplo completo

```
version: '3.9'
services:
  gateway:
    build: ./gateway
    ports:
      - "8080:8080"
    depends_on:
      - usuarios
    networks:
      - backend

  usuarios:
    image: utnfc/ms-usuarios:1.0
    networks:
      - backend

networks:
  backend:

volumes:
  db-data:
```

> Este archivo levanta dos servicios que comparten una red, uno de los cuales se construye localmente y el otro se descarga como imagen, ambos gestionados en conjunto.

En los siguientes apartados vamos a profundizar en el uso de **volúmenes** y **redes**, conceptos clave para garantizar persistencia de datos, seguridad y una arquitectura limpia entre microservicios.

Volúmenes en Docker

Los contenedores Docker son, por diseño, efímeros. Esto significa que cualquier archivo escrito en el sistema de archivos de un contenedor se pierde una vez que el contenedor se detiene o elimina. En este contexto, los **volúmenes** se presentan como una solución fundamental para garantizar la **persistencia de datos**.

Un volumen en Docker es una sección del sistema de archivos del **host** (o sistema de archivos anfitrión) que se monta dentro del contenedor. Esta estrategia permite que los datos sobrevivan al ciclo de vida del contenedor, habilita el intercambio de archivos entre contenedores y mejora el rendimiento en ciertos entornos.

Fundamento Conceptual

Cuando creamos un contenedor, por ejemplo con `docker run`, si ese contenedor genera archivos en su sistema de archivos interno, dichos archivos vivirán dentro de su estructura **UnionFS** (el sistema de archivos por capas). Sin embargo, esa información no estará disponible si el contenedor se elimina o si otro contenedor requiere acceso a los mismos datos.

Comparativa: Volúmenes Docker vs Montajes NFS

Los volúmenes son la respuesta de los contenedores docker a lo que haríamos con NFS (Network File System) si estuvieramos trabajando con equipos físicos o máquinas virtuales, la diferencia es que el UnionFS al ser declarativo y convivir en el mismo quernel lo vuelve natural.

Además nada impide que las estrategias de Docker se complementen con estrategias NFS para montar los archivos reales en un Storage separado del SO anfitrión por ejemplo:

Característica	Volúmenes Docker	Montajes NFS
Creación y gestión	Integrada con Docker CLI	Requiere configuración externa y permisos
Persistencia	Persisten tras eliminación del contenedor	Igual, si está correctamente montado
Ubicación	Dentro del área de gestión de Docker	Carpeta específica en el sistema anfitrión
Portabilidad	Fácil con Docker Compose y backups	Requiere montar manualmente en cada host
Rendimiento	Rápido (local al host)	Más lento por depender de red
Acceso concurrente entre hosts	No (por defecto)	Sí (diseñado para entornos distribuidos)
Seguridad	Controlada por Docker	Depende de permisos de sistema y red
Requiere configuración extra?	No	Sí (exports, permisos, montaje en cada host)
Ideal para...	Apps en un solo host con Docker	Clústeres o entornos multi-host compartidos

Los volúmenes permiten salvar esa barrera, al montar directorios del sistema anfitrión en puntos específicos del sistema de archivos del contenedor.

Esto los hace imprescindibles para:

- Bases de datos (persistencia de datos entre reinicios)
- Aplicaciones que generan o consumen archivos
- Inicialización de datos desde el host
- Compartir archivos entre contenedores

Creación y gestión de volúmenes

Comando	Descripción
`docker volume create NOMBRE`	Crea un nuevo volumen llamado `NOMBRE`
`docker volume ls`	Lista todos los volúmenes existentes
`docker volume inspect NOMBRE`	Muestra detalles de un volumen (ruta en el host, uso, etc.)
`docker volume rm NOMBRE`	Elimina un volumen no utilizado por ningún contenedor

> ⚠ **Nota**: No se puede eliminar un volumen que esté montado en un contenedor en ejecución o detenido. Primero hay que eliminar el contenedor o desmontar el volumen.

Usando volúmenes en `docker run`

Podemos montar un volumen al crear un contenedor:

```
docker run -v mi-volumen:/app/data imagen
```

- `mi-volumen` es el nombre del volumen
- `/app/data` es el punto de montaje dentro del contenedor

> En teste caso el volumen `mi-volumen` debe existir previamente es decir debe haber sido creado con `docker. volume` antes de ser utilizado y puede ser utilizado por varios contenedores a la vez.

También podemos usar rutas absolutas del sistema operativo anfitrión:

```
docker run -v /home/feli/archivos:/data imagen
```

Esto monta la carpeta `/home/feli/archivos` del host en `/data` dentro del contenedor.

> En este caso no se está creando formalmente un volumen y por lo tanto si bien varios contenedores pueden apuntar al mismo directorio físico del sistema de archivos no queda formalizado como un volumen docker.

Volúmenes en Docker Compose

Compose permite una declaración mucho más organizada de los volúmenes. Veamos el ejemplo de un contenedor PostgreSQL:

```
```

```
services:
  postgres:
    image: postgres:${POSTGRES_VERSION}
    container_name: postgres
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      - ./pgdata:/var/lib/postgresql/data
      - ./initdb:/docker-entrypoint-initdb.d
```

Aquí tenemos dos usos complementarios:

1. `./pgdata:/var/lib/postgresql/data` → Mapea una carpeta del host (`pgdata`) donde se almacenarán los datos persistentes.
2. `./initdb:/docker-entrypoint-initdb.d` → Mapea una carpeta con scripts de inicialización que serán ejecutados al crear el contenedor por primera vez.

Evolución del uso de volúmenes

A medida que complejizamos nuestras soluciones, los volúmenes pueden tomar distintas formas:

Tipo de volumen	Declaración	Ejemplo
Volumen nombrado	`nombre:/path`	`dbdata:/var/lib/mysql`
Bind mount (ruta absoluta)	`/host/path:/path`	`/home/feli/proyecto:/app`
Solo destino (volumen anónimo)	`/path`	`/data`

En Docker Compose también podemos definir volúmenes al final del archivo:

```
volumes:
  dbdata:
```

Esto ayuda a mantener una configuración más clara y reutilizable entre servicios.

En este caso docker crea el volumen y lo guarda en un directorio específico configurado para almacenar volúmenes nombrados, en un sistema anfitrión linux estándar será en:

```
/var/lib/docker/volumes/dbdata/_data
```

> [!WARNING]
>
> Al usar volúmenes nombrados hay que tener en cuenta que los nombres se van a compartir entre todos los contenedores que residan en el mismo sistema operativo, sean del mismo compose

> (backend) o no, y por ello debemos ser descriptivos y unívocos al establecer los nombres (`dbdata` no sería un buen nombre ya que si se monta más de una base de datos en el host podría generar confusión, un nombre mejor sería por ejemplo 'postgres-data')

Si creamos previamente los volúmenes para nuestro compose, el archivo `docker-compose.yml` podría evolucionar entonces a:

```
services:
  postgres:
    image: postgres:${POSTGRES_VERSION}
    container_name: postgres
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      - postgres-data:/var/lib/postgresql/data
      - postgres-initdb:/docker-entrypoint-initdb.d

volumes:
  postgres-data:
  postgres-initdb:
```

En los siguientes apartados profundizaremos en los distintos tipos de volúmenes, sus ventajas, y las buenas prácticas para organizarlos en proyectos reales.

🔷 En resumen: Tipos de Volúmenes

1. Volúmenes Nombrados (Named Volumes)

```
volumes:
  - postgres-data:/var/lib/postgresql/data

volumes:
  postgres-data:
```

- **Gestión**: Docker los gestiona internamente.
- **Persistencia**: Persisten aunque se destruya el contenedor.
- **Ubicación**: En `/var/lib/docker/volumes/<nombre>/_data`.
- **Portabilidad**: Ideales para entornos productivos o consistentes.
- **Backup**: Fácil de respaldar.

Uso recomendado: cuando el volumen no necesita estar en una ruta específica del host y se busca aislamiento.

2. Bind Mounts (Montajes Directos)

```
volumes:
  - ./pgdata:/var/lib/postgresql/data
```

- **Gestión**: El usuario controla la ruta del host.
- **Persistencia**: Persisten en el path especificado.
- **Ubicación**: Ruta definida explícitamente por el usuario.
- **Desventajas**: Más frágil a cambios de estructura o permisos.
- **Portabilidad**: No ideal para compartir entre entornos.

Uso recomendado: desarrollo, pruebas locales o cuando se requiere control total sobre los archivos del host.

3. Volumes Anónimos

```
docker run –v /data busybox
```

- **Nombre**: Docker crea un nombre aleatorio.
- **Ubicación**: Docker lo gestiona como un volumen nombrado.
- **Problema**: Más difícil de rastrear y reutilizar.

Uso recomendado: rara vez, o en scripts desechables de testing.

🧭 Buenas Prácticas

Situación	Tipo de Volumen Recomendado	Justificación
Producción (Postgres, App State)	Volumen Nombrado	Aislado, controlado por Docker, portátil
Desarrollo local con necesidad de editar archivos	Bind Mount	Permite ver y modificar cambios en tiempo real
Testing automatizado, sin persistencia	Anónimo o sin volumen	No requiere retención de datos
Multiples contenedores compartiendo datos	Nombrado o Bind Mount (según control requerido)	Asegura sincronía y persistencia

📝 Recomendaciones Generales

- Usar **volúmenes nombrados** por defecto, salvo que se requiera observar/modificar los datos desde el host.
- Evitar bind mounts en producción salvo casos puntuales (certificados, logs, etc).
- Siempre documentar el uso del volumen: origen, propósito, ubicación esperada.
- Nunca montar directorios críticos del host por accidente (`/`, `/etc`, etc).
- Utilizar `.dockerignore` para evitar copiar archivos innecesarios que luego aparezcan en volúmenes mal definidos.

Redes en Docker

Conceptos generales

Docker nos permite definir y gestionar redes virtuales que conectan contenedores entre sí de forma aislada y controlada. Así como los volúmenes resuelven la persistencia de datos, las redes en Docker resuelven la comunicación entre servicios sin depender del entorno físico o la configuración del host.

Esto nos permite crear un entorno de microservicios donde los contenedores se pueden comunicar por nombre de servicio, sin necesidad de exponer puertos innecesariamente al exterior, favoreciendo la seguridad y escalabilidad del sistema.

¿Por qué redes?

Cuando usamos `docker run` sin definir una red, Docker automáticamente conecta los contenedores a una red por defecto llamada `bridge`. Esta red es suficiente para contenedores individuales o pruebas simples, pero no nos brinda el control necesario para arquitecturas complejas.

Al trabajar con múltiples contenedores que forman parte de un sistema (por ejemplo, un backend compuesto por microservicios y un gateway), necesitamos que:

- Los servicios se comuniquen internamente sin exponer sus puertos al host.
- Solo el gateway esté expuesto al exterior.
- Pueda haber múltiples entornos aislados (dev, test, prod) en paralelo.

Esto lo logramos declarando redes explícitas en `docker-compose.yml`.

Redes por nombre y resolución DNS interna

Cuando dos servicios están en la misma red definida por Compose, pueden comunicarse directamente usando el nombre del servicio como hostname. Por ejemplo, si tenemos:

```
services:
  ms-usuarios:
    image: miapp/usuarios

  ms-tarifas:
    image: miapp/tarifas
```

Entonces `ms-usuarios` puede hacer una petición a `http://ms-tarifas:8080/api/tarifas` sin necesidad de conocer la IP.

Definición de redes en Compose

Las redes se definen al final del archivo `docker-compose.yml` y luego se asignan a cada servicio:

```
services:
  gateway:
```

```
    image: miapp/gateway
    ports:
      - "8080:8080"
    networks:
      - red-backend

  ms-usuarios:
    image: miapp/usuarios
    networks:
      - red-backend

  ms-tarifas:
    image: miapp/tarifas
    networks:
      - red-backend

networks:
  red-backend:
```

> ⚠ Como no definimos opciones en `red-backend`, Compose la crea como red `bridge` aislada automáticamente.

Control de exposición externa

Para decidir qué servicios son accesibles desde fuera del contenedor, usamos la opción `ports:`.

En desarrollo

Podemos exponer todos los microservicios para acceder directamente desde Postman, navegador, etc.:

```
services:
  ms-usuarios:
    ports:
      - "8081:8080"
  ms-tarifas:
    ports:
      - "8082:8080"
```

En test/preproducción/producción

Podemos dejar expuesto únicamente el API Gateway:

```
services:
  gateway:
    ports:
      - "8080:8080"
  ms-usuarios:
    expose:
```

```
      - "8080"
  ms-tarifas:
    expose:
      - "8080"
```

- `ports` hace un **bind** del puerto del contenedor al host.
- `expose` documenta el puerto disponible dentro de la red **sin exponerlo al host**.

Esto nos permite aislar los servicios internos, aumentando la seguridad y alineándonos con el patrón API Gateway.

Tabla Comparativa: Red `bridge` vs Red definida

Característica	**Red `bridge` por defecto**	**Red definida en Compose**
Comunicación entre contenedores	No garantizada	Por nombre de servicio
Aislamiento	Bajo	Alto
Control de exposición	Limitado	Total
Escalabilidad	Manual	Simplificada
Reutilización	Baja	Alta

Buenas Prácticas

- **Nombrar las redes** explícitamente para facilitar el mantenimiento.
- **Usar una única red por entorno** para aislar servicios.
- **No exponer puertos innecesarios**: exponer solo el gateway o API pública.
- **Documentar las redes** y su propósito dentro del `docker-compose.yml`.
- **Combinar con variables de entorno** para que los servicios usen los nombres correctos al resolver URLs.

Laboratorio 1 - Despliegue de un backend con Docker Compose

En este laboratorio vamos a consolidar todos los conocimientos adquiridos sobre Docker y su herramienta Compose, desplegando un backend completo compuesto por tres microservicios. A diferencia de los laboratorios anteriores, el objetivo aquí no es estudiar cada comando de Docker de forma aislada, sino comprender cómo orquestar múltiples servicios que se comunican entre sí, utilizando una red compartida y exponiendo únicamente la interfaz del Gateway para su consumo externo.

Objetivo General

> Desplegar una arquitectura de microservicios Java Spring Boot utilizando Docker Compose, dejando expuesto únicamente el Gateway para su uso externo y permitiendo que el resto de los microservicios funcionen dentro de la red de Docker.

Estructura del Proyecto

Vamos a partir del Backend generado para la ejemplificación del Bloque 9 de contenidos que modela un servicio para adminstrar comidas y otro para sugerir maridajes de acuerdo con la comida especificada, le hemos agregado un gateway absolutamente básico que solo configura los mapeos y no hace absolutamente nada más que servir como puerta de entrada al backend.

El backend está conformado por los siguientes proyectos:

- `ms-comidas`: Servicio funcional que gestiona un listado de comidas.
- `ms-maridaje`: Servicio funcional que recibe comidas y responde con sugerencias de bebidas asociadas.
- `gateway`: Servicio que actúa como entrada única (API Gateway) al backend.

Cada uno de estos proyectos está ubicado en su propio subdirectorio y posee su respectivo `Dockerfile` para construir la imagen del servicio.

Organización en el sistema de archivos

```
foodmatch-backend/
├── gateway/         --> Servicio de API Gateway (Spring Cloud Gateway)
├── ms-comidas/      --> Microservicio que expone comidas posibles
├── ms-maridaje/     --> Microservicio que expone combinaciones posibles
entre comidas y bebidas
```

Cada uno de estos proyectos es un servicio independiente, empaquetado como aplicación Spring Boot con su propio archivo `pom.xml`, sus controladores y su configuración de puertos para ejecución local o en contenedor.

Requisitos

- Java 21
- Maven 3.9+
- Docker y Docker Compose
- (Opcional) Extensión REST Client en VS Code para realizar pruebas

Paso 1 - Verificación local sin Docker

Antes de utilizar Docker, es fundamental verificar que la aplicación funciona correctamente al ejecutarla directamente desde el entorno de desarrollo:

```
cd ms-comidas && mvn spring-boot:run
cd ms-maridaje && mvn spring-boot:run
cd gateway && mvn spring-boot:run
```

Los puertos por defecto en esta ejecución local son:

- `ms-comidas`: 8081
- `ms-maridaje`: 8082

- `gateway`: 8080

Se recomienda iniciar los microservicios en orden de dependencias: primero `ms-comidas`, luego `ms-maridaje`, y finalmente el `gateway`.

Paso 2 - Modificaciones necesarias en `application.yml`

Al desplegar con Docker, los microservicios no deben comunicarse utilizando `localhost`, sino los nombres de los contenedores dentro de la red de Docker.

ms-maridaje/src/main/resources/application.yml

```
comidas:
  base-url: http://ms-comidas:8081/api/comidas
```

gateway/src/main/resources/application.yml

```
server:
  port: 8080

spring:
  cloud:
    gateway:
      routes:
        - id: comidas
          uri: http://ms-comidas:8081
          predicates:
            - Path=/api/comidas/**
        - id: maridaje
          uri: http://ms-maridaje:8082
          predicates:
            - Path=/api/maridaje/**
```

> Recordemos que dentro de la red Docker, `ms-comidas` y `ms-maridaje` son nombres de host válidos.

Paso 3 - Construcción de imágenes

Cada proyecto tiene un `Dockerfile` con el siguiente contenido base:

```
FROM openjdk:21
WORKDIR /app
COPY target/app.jar app.jar
EXPOSE 808X
ENTRYPOINT ["java", "-jar", "app.jar"]
```

> [!TIP] Tener en cuenta de hacer referencia correcta al nobmre de cada archivo `.jar` en el commando `COPY` de cada `Dockerfile`

Se debe compilar cada proyecto antes de construir la imagen:

```
cd ms-comidas && mvn clean package
cd ms-maridaje && mvn clean package
cd gateway && mvn clean package
```

Paso 4 - Definición de los archivos Compose

El archivo `docker-compose.yml` se ubica en la raíz del proyecto, permitiendo orquestar todos los servicios de manera centralizada.

Algunas buenas prácticas para Docker Compose con múltiples microservicios

1. **Un `docker-compose.yml` en la raíz**
    - Facilita levantar todo el backend con un solo comando.
    - Permite definir variables compartidas en `.env`.
2. **Cada microservicio con su propio `Dockerfile`**
    - Ubicado dentro de su carpeta respectiva (ej: `ms-comidas/Dockerfile`).
    - Asegura independencia y claridad en el proceso de build.
3. **Utilizar `build.context` y `build.dockerfile`**
    - Evita ambigüedades al indicar desde dónde construir la imagen.

    ```
    services:
      ms-comidas:
        build:
          context: ./ms-comidas
          dockerfile: Dockerfile
    ```
4. **Nombres de servicios claros y coherentes**
    - Ej: `ms-comidas`, `ms-maridaje`, `gateway`
5. **Asignar puertos solo al gateway en entorno de test/prod**
    - En desarrollo podrías exponer todos para debugging, pero en otros entornos, sólo se expone el gateway.
6. **Definir redes y volúmenes al final del archivo**
    - Mantiene la composición ordenada y reutilizable.

7. **Usar `.env` para parametrizar variables**
    - Versionar `.env.example`, no el `.env` real.
    - Vamos a profundizar sobre este tema en el próximo bloque de contenidos

Alternativas de Composición

Opcion A: Compose con `Dockerfile` por microservicio (recomendado)

Ventajas:

- Modularidad: cada servicio puede evolucionar su `Dockerfile` independientemente.
- Reutilizable: cada servicio puede desplegarse aislado si se requiere.

Desventajas:

- Requiere mantener y versionar múltiples `Dockerfile`.

Opcion B: Compose con una sola imagen preconstruida por servicio

```
services:
  ms-comidas:
    image: utnfc/ms-comidas:1.0
```

Ventajas:

- Más rápido al no requerir build local.
- útil para ambientes de test o producción.

Desventajas:

- Requiere un paso previo de build/push.
- Menos flexible para iteración local.

Archivos `docker-compose.yml`

Al trabajar con múltiples microservicios que conforman el backend de una aplicación, es habitual que necesitemos distintos comportamientos según el entorno en el que estemos trabajando. Por ejemplo:

- En **entornos de desarrollo**, suele ser útil exponer todos los microservicios funcionales directamente, para poder accederlos fácilmente desde herramientas como Postman o REST Client.
- En **entornos de prueba, preproducción o producción**, en cambio, queremos que solamente el **API Gateway** sea accesible desde el exterior, y que los demás microservicios estén aislados dentro de la red interna del backend.

Para lograr este comportamiento, una buena práctica es separar la configuración de Docker Compose en **dos archivos independientes**:

1. `docker-compose.yml`

Este archivo contiene la configuración base y común para todos los entornos. Define los servicios, sus puertos internos, dependencias y red compartida, pero **no publica puertos externos de los microservicios funcionales** (solo el gateway lo hace).

2. `docker-compose.override.yml`

   Este archivo se utiliza únicamente en desarrollo y complementa al archivo principal, agregando mapeos de puertos adicionales para exponer directamente los microservicios funcionales. Docker Compose lo aplica automáticamente al ejecutar `docker compose up` sin ningún parámetro adicional.

   > [!NOTE]
   > Esta estrategia permite mantener una sola fuente de verdad (`docker-compose.yml`) para todos los entornos, mientras que se personaliza el comportamiento de desarrollo sin alterar la configuración base. Esto evita errores por exponer puertos innecesarios en ambientes donde no corresponde.

Los archivos deben ubicarse en la carpeta raíz del proyecto `foodmatch-backend/` y contienen los tres servicios, una red personalizada. Y luego la redefinicion para que se expongan los 3 servicios para desarrollo.

📁 **docker-compose.yml**

```
version: '3.9'

services:

  ms-comidas:
    build:
      context: ./ms-comidas
    container_name: ms-comidas
    networks:
      - foodmatch-net

  ms-maridaje:
    build:
      context: ./ms-maridaje
    container_name: ms-maridaje
    networks:
      - foodmatch-net
    depends_on:
      - ms-comidas

  gateway:
    build:
      context: ./gateway
    container_name: gateway
        ports:
      - "9000:8080"
    networks:
      - foodmatch-net
```

```
    depends_on:
      - ms-comidas
      - ms-maridaje

networks:
  foodmatch-net:
    driver: bridge
```

> Este archivo **no publica los puerto de los microservicios funcionales**, por lo tanto ninguno de los servicios funcionales es accesible desde fuera de Docker por defecto, **solo publica el puerto del gateway** y por lo tanto lo único accesible desde fuera será el puerto del gateway.

📁 `docker-compose.override.yml` para desarrollo

Este archivo redefine los puertos expuestos para los entornos de desarrollo.

```
version: '3.9'

services:
  ms-comidas:
    ports:
      - "8081:8081"

  ms-maridaje:
    ports:
      - "8082:8082"

  gateway:
    ports:
      - "9001:8080"
```

Ejecuciones según entorno

Desarrollo (modo completo: todos los servicios expuestos)

```
docker compose up --build
```

> Esto utilizará **ambos archivos** (`docker-compose.yml` + `docker-compose.override.yml`)

Producción (modo gateway solamente expuesto)

```
docker compose --file docker-compose.yml up --build
```

> Esto ignora el archivo override y mantiene **todos los servicios funcionales no expuestos** excepto si el Dockerfile del gateway ya lo hace explícitamente.

Otra alternativa

Otra alternativa es hacerlo directamente con dos archivos `docker-compose.xxx.yml` diferentes de forma que usemos el que necesitemos en cada caso, esta última es quizás la mejor opción cuando queremos realizar configuraciones epecíficas para cada entorno.

- `docker-compose.dev.yml`: expone todos los puertos.
- `docker-compose.prod.yml`: expone solo el gateway.

Paso 5 - Ejecución

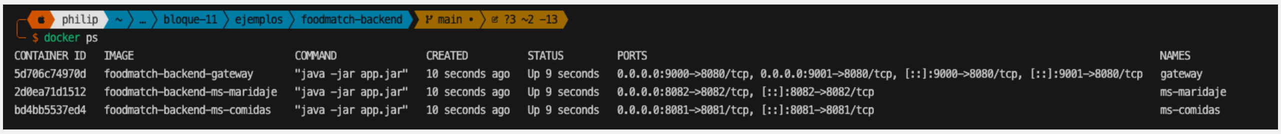
Desarrollo

Por defecto compose utiliza todos los archivos disponibles por lo que el archivo `docker-compose.override.yml` redefine los valores de `docker-compose.yml` y expone todos los servicios.

```
docker compose up --build -d
```

En otra terminal se puede comprobar mediante:

```
docker ps
```

> Deberíamos observar el siguiente resultado:
> 

Este backend nos permite acceder a:

- http://localhost:8081/api/comidas
- http://localhost:8082/api/maridaje
- http://localhost:9001/api/comidas

Desarrollo (se expone cada microservicio)

Para un despliegue de producción donde cerremos los puertos de los micros servicios funcionales podemos hacerlo mediante:

```
docker compose --file docker-compose.yml up --build -d
```

> Aunque también podríamos no tener directamente el archivo `docker-compose.override.yml`, no mergeando este archivo a la rama de producción por ejemplo.

También podemos comprobarlo con el comando `docker ps` y solo nos permitirá acceso al puerto 9000

Acceder a: http://localhost:9000/api/comidas

Referencia

El presente laboratorio puede analizarse y ponerse en marcha desde el ejemplo Food Match Backend

Conclusiones

Este laboratorio nos permitió:

- Consolidar el conocimiento sobre `Dockerfile`.
- Comprender cómo Compose automatiza la construcción y orquestación de múltiples contenedores.
- Ver la importancia de las redes de Docker para la comunicación interna.
- Experimentar con diferentes estrategias de exposición de puertos para entornos de desarrollo y producción.

El uso de Docker Compose marca un antes y un después en la forma de trabajar con múltiples servicios, simplificando enormemente el despliegue, prueba y mantenimiento del backend completo.

> ¡A partir de ahora, nuestro backend puede funcionar como un bloque único y portátil! ✨

prendiendo

- **Docker Compose**
  Manual de referencia para la orquestación de múltiples contenedores usando archivos YAML.
  👉 https://docs.docker.com/compose/